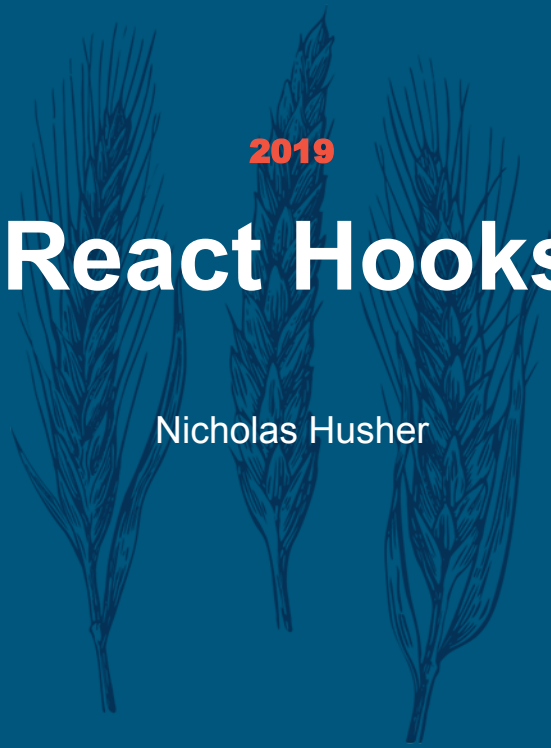


2019

React Hooks

Nicholas Husher

indigo™

Outline

- Zeroth: What are hooks?
- First: Overview of standard React hooks
- Second: Some custom hook cookbooks

How are hooks different?

- Makes components smaller by splitting logic from components
- Reduces duplication by sharing logic between components
- Simplifies large problems by decomposing them into smaller, simpler pieces
- Reduces complicated patterns by offering a single clear way to compose logic

Part 1

Your new toolbox

indigo™

useState

```
function ColorPicker () {
  const [color, setColor] = useState('white');


  return <>
    <div style={{ color }}>
      The color is {color}
    </div>

    <button onClick={() => setColor('red')}>red</button>
    <button onClick={() => setColor('green')}>green</button>

  </>
}
```

useState

```
function ColorPicker () {
  const [color, setColor] = useState('white');
  const [count, setCount] = useState(0);

  return <>
    <div style={{ color }}>
      The color is {color} and the count is {count}
    </div>

    <button onClick={() => setColor('red')}>red</button>
    <button onClick={() => setColor('green')}>green</button>
    <button onClick={() => setCount(v => v + 1)}>increment</button>
  </>
}
```

useEffect

```
function User () {
  useEffect(() => {
    fetch('/user')
      .then(res => res.json())
      .then(console.log)
  }, [])

  return <div></div>
}
```

useEffect

```
function User () {
  const [user, setUser] = useState(null)
  useEffect(() => {
    fetch('/user').then(res => res.json()).then(setUser)
  }, [])

  if (!user) return <div>loading...</div>
  return <div>{user.name} ({user.email})</div>
}
```

useEffect

```
function User ({ id }) {
  const [user, setUser] = useState(null)
  useEffect(() => {
    fetch(`/user/${id}`).then(res => res.json()).then(setUser)
  }, [])

  if (!user) return <div>loading...</div>
  return <div>{user.name} ({user.email})</div>
}
```

useEffect

```
function User ({ id }) {
  const [user, setUser] = useState(null)
  useEffect(() => {
    fetch(`/users/${id}`).then(res => res.json()).then(setUser)
  }, [id])

  if (!user) return <div>loading...</div>
  return <div>{user.name} ({user.email})</div>
}
```

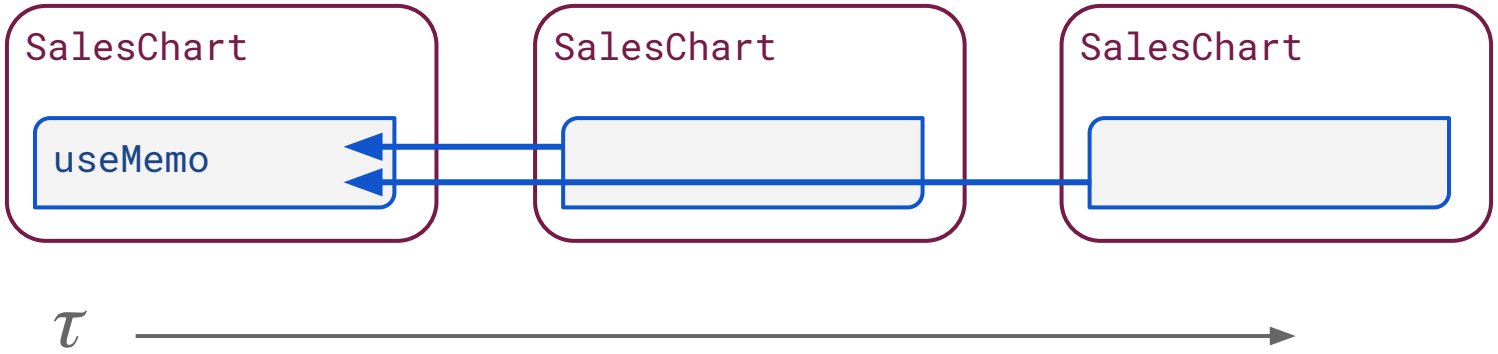
useEffect

```
function Counter () {
  const [count, setCount] = useState(0)

  useEffect(() => {
    const interval = setInterval(() => setState(c => c +  1), 1000)

    return () => clearInterval(interval)
  }, [])

  return <div>{count}</div>
}
```

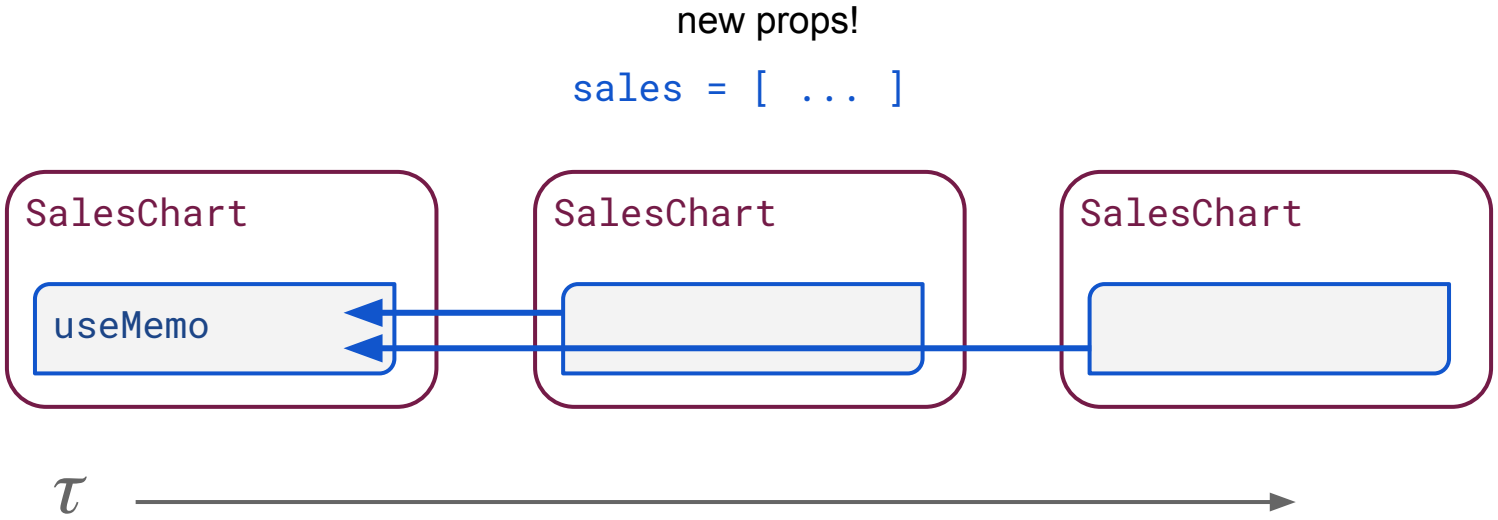
useMemo

```
// sales = [{ id: 1, amount: 50.00, date: new Date(2019, 10, 1) }, ...]

function SalesChart ({ sales }) {
  const aggregated = useMemo(() =>
    sales.reduce(aggregateSalesByMonth, []),
    [])

  return <VictoryBar data={aggregated} ... />
}
```

useMemo



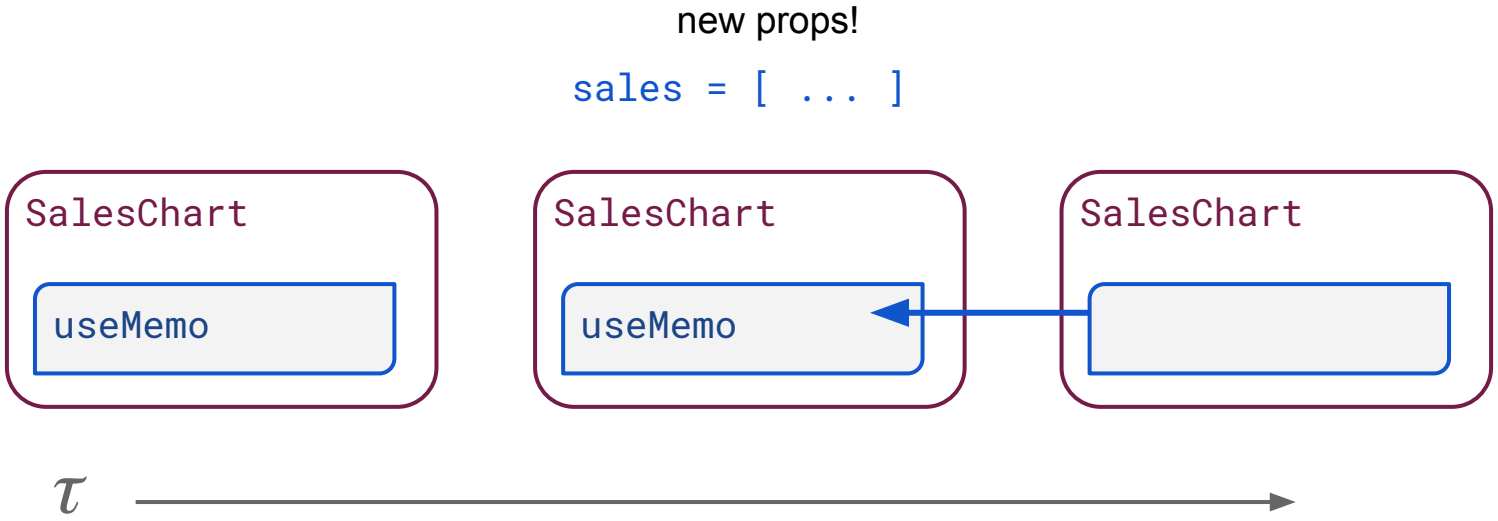
useMemo



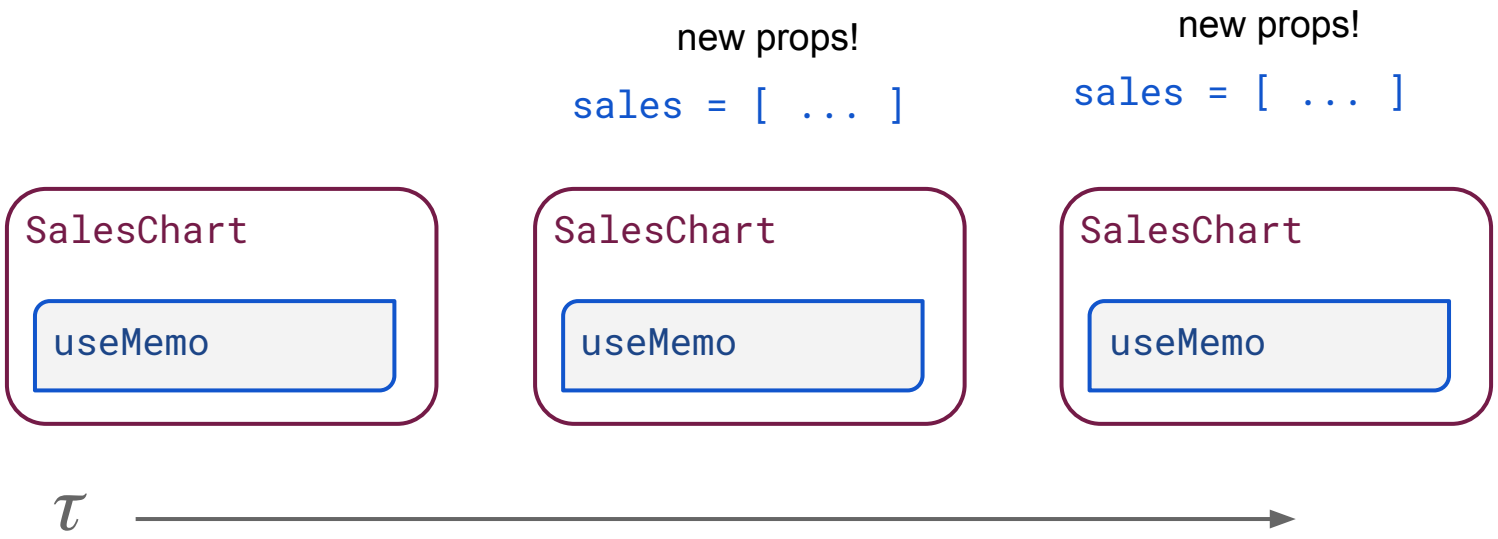
useMemo

```
// sales = [{ id: 1, amount: 50.00, date: new Date(2019, 10, 1) }, ...]

function SalesChart ({ sales }) {
  const aggregated = useMemo(() =>
    sales.reduce(aggregateSalesByMonth, []),
    [sales])

  return <VictoryBar data={aggregated} ... />
}
```

useMemo


new props!
sales = [...]
SalesChart
useMemo
SalesChart
useMemo
SalesChart
τ

useMemo



useQuery

```
function User ({ id }) {
  return <Query query={query} variables={{ id }}>{({ data, loading }) => {
    if (loading) return <div>loading...</div>
    return <div>{data.user.name}</div>
  }}</Query>
}

const query = gql`
  query get_user (id: ID!) {
    user: get_user (id: $id) {
      name, email
    }
  }
`
```

useQuery

```
function User ({ id }) {
  const { data, loading } = useQuery(query, { variables: { id }})

  if (loading) return <div>loading...</div>
  return <div>{data.user.name}</div>
}

const query = gql`
  query get_user (id: ID!) {
    user: get_user (id: $id) {
      name, email
    }
  }
`
```

useQuery

```
function User ({ id }) {
  return <Query query={userQuery}>{({ data: userData, loading: userLoading }) =>
    <Query query={accountQuery}>{({ data: accountData, loading: accountLoading }) =>
      <Mutation mutation={userMutation}>{(updateUser) => {
        if (userLoading) return <div>loading...</div>

        return <div>{userData.user.name} ({accountData.account.name})</div>
      }}</Mutation>
    </Query>
  </Query>
}
```

useQuery

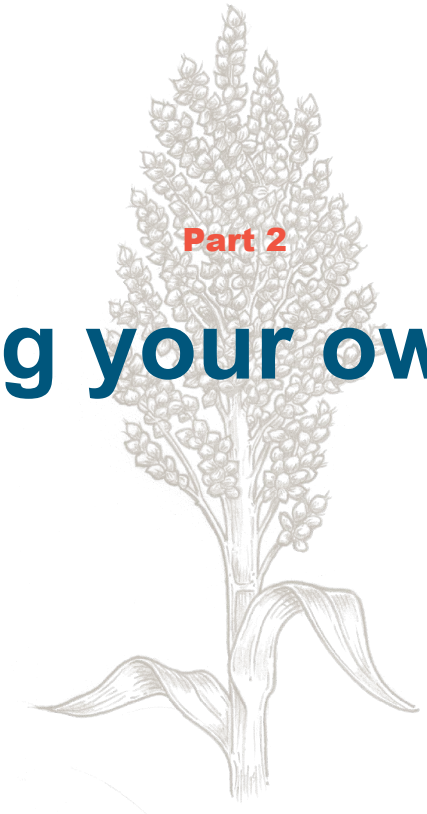
```
function User ({ id }) {
  const { data: userData, loading: userLoading } = useQuery(userQuery)
  const { data: accountData, loading: accountLoading } = useQuery(accountQuery)
  const [updateUser] = useMutation(userMutation)

  if (userLoading) return <div>loading...</div>

  return <div>{userData.user.name} ({accountData.account.name})</div>
}
```

Some others

- `useContext` — For importing Context values without a render prop
- `useRef` — For storing DOM node references and other things
- `useReducer` — Like setState but for more complicated state
- `useCallback` — Retain function identities between renders
- `useImperativeHandle`, `useLayoutEffect`, `useDebugValue`

Important witchcraft

- Always call the same hooks on every render, in the same order
- Data dependencies must be accurate
- Beware stale closures
- Hooks beget hooks, so use utility functions!



Part 2

Forging your own tools

indigo™

Example: Tracking mouse position

```
let listeners = [];
document.addEventListener('mousemove', e => listeners.forEach(l => l(e)))

function useMousePosition () {
  const [position, setPosition] = useState({ x: null, y: null })

  useEffect(() => {
    const listener = ({ pageX: x, pageY: y }) => setPosition({ x, y })
    listeners.push(listener)

    return () => {
      listeners = listeners.filter(l => l !== listener)
    }
  })

  return position
}
```

Example: Tracking mouse position

```
function MouseTracker () {
  const { x, y } = useMousePosition()

  return <div>The mouse is at {x}x {y}y</div>
}
```

Example: Redux lite

```
const Store = React.createContext();

function StoreProvider ({ children, reducer, initial }) {
  const [ state, dispatch ] = useReducer(reducer, initial)
  const store = { state, dispatch }

  return <Store.Provider value={store}>{children}</Store.Provider>
}

function connect (mapStateToProps, mapDispatchToProps) {
  return function Connected (Component) {
    const { state, dispatch } = useContext(Store)
    const mappedState = mapStateToProps(state)
    const mappedDispatch = mapDispatchToProps(dispatch)

    return <Component {...mappedState} {...mappedDispatch} />
  }
}
```

Example: Redux lite

```
function countingReducer (state, { type }) {
  if (type === 'INCREMENT') return state + 1
  else if (type === 'DECREMENT') return state - 1
  else return state
}

function App () {
  return <StoreProvider reducer={countingReducer} initial={0}>
    <UIContainer />
  </StoreProvider>
}
```

Example: Redux lite

```
function mapDipatchToProps (dispatch) {
  return {
    increment() { dispatch({ type: 'INCREMENT' })},
    decrement() { dispatch({ type: 'DECREMENT' })},
  }
}

function UI({ count, increment, decrement }) {
  return <div>
    <h1>{count}</h1>
    <button onClick={increment}>+</button>
    <button onClick={decrement}>-</button>
  </div>
}

const UIContainer = connect(count => ({ count }), mapDispatchToProps)(UI)
```

Example: WebSocket hook

```
function useWebSocket (url, onMessage) {
  const messageRef = useRef();
  const socketRef = useRef();

  useEffect(() => {
    const socket = new WebSocket(url)
    socket.onmessage = e => messageRef.current(JSON.parse(e.data))
    socketRef.current = socket

    return () => socket.close()
  }, [url])

  useEffect(() => { messageRef.current = onMessage }, [onMessage])

  return useCallback(data => socketRef.current.send(JSON.stringify(data)), [])
}
```

Example: WebSocket hook

```
function Chat () {
  const chatBoxRef = useRef()
  const [ messages, setMessages ] = useState([])
  const send = useWebSocket(chatUrl, message => {
    setMessages(m => m.concat(message))
  })

  return <div>
    {messages.map(m => <p>{m}</p>)}
    <input type="text" ref={chatBoxRef} />
    <button onClick={() => send(chatBoxRef.current.value)}>send</button>
  </div>
}
```

Takeaways

- Hooks can make your code smaller
  - No more nested render prop components!
  - No more splitting logic across many lifecycle hooks
- You can use them to separate logic into reusable parts
  - Simpler units of code are easier to understand
  - Simple units of code can be composed to make complex features
- Create hooks by composing built-in React hooks
- But there are some rules that you must follow
  - Always call every hook every time
  - Watch out for stale closures

Questions?

indigo™